Sciduction: Combining Induction, Deduction, and Structure for Verification and Synthesis

Sanjit A. Seshia

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-68
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-68.html

May 26, 2011

Sciduction: Combining Induction, Deduction, and Structure for Verification and Synthesis

Sanjit A. Seshia
University of California, Berkeley
sseshia@eecs.berkeley.edu

Abstract

Even with impressive advances in automated formal methods, certain problems in system verification and synthesis remain challenging. Examples include the verification of quantitative properties of software involving constraints on timing and energy consumption, and the automatic synthesis of systems from specifications. The major challenges include environment modeling, incompleteness in specifications, and the complexity of underlying decision problems.

This position paper proposes sciduction, an approach to tackle these challenges by integrating *inductive inference*, *deductive reasoning*, and *structure hypotheses*. Deductive reasoning, which leads from general rules or concepts to conclusions about specific problem instances, includes techniques such as theorem proving and constraint solving. Inductive inference, which generalizes from specific instances to yield a concept, includes algorithmic learning from examples. Structure hypotheses are used to define the class of artifacts, such as invariants or program fragments, generated during verification or synthesis. Sciduction constrains inductive and deductive reasoning using structure hypotheses, and actively combines inductive and deductive reasoning: for instance, deductive techniques generate examples for learning, and inductive reasoning is used to guide the deductive engines.

We illustrate this approach with three applications: (i) timing analysis of software; (ii) synthesis of loop-free programs, and (iii) controller synthesis for hybrid systems. Some future applications are also discussed.

1 Introduction

Formal verification has made enormous strides over the last few decades. Verification techniques such as model checking [11, 41, 13] and theorem proving (see, e.g. [22]) are used routinely in computer-aided design of integrated circuits and have been widely applied to find bugs in software. However, certain problems in system verification and synthesis remain very challenging. This position paper seeks to outline these challenges and presents an approach for tackling them along with some initial evidence for the utility of the approach.

The traditional view of verification is as a decision problem, with three inputs (see Figure 1):

1. A model of the system to be verified, S;
2. A model of the environment, E, and
3. The property to be verified, Φ.

The verifier generates as output a YES/NO answer, indicating whether or not S satisfies the property Φ in environment E. Typically, a NO output is accompanied by a counterexample, also called an error trace, which is an execution of the system that indicates how Φ is violated. Some formal verification tools also include a proof or certificate of correctness with a YES answer.

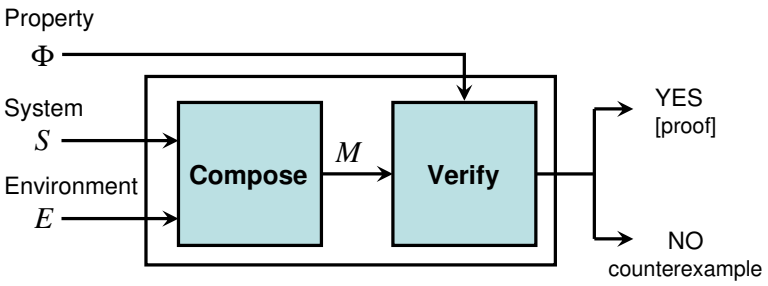

Figure 1: Formal verification procedure.

While this view of verification has proved effective for many problems, it is somewhat idealized. In practice, one does not always start with models S and E — these might have to be generated from implementations. In the case of the system model S, techniques for automatic abstraction have proved to be effective in practice (e.g., the use of predicate abstraction in software model checking [5]). However, generating an adequately precise environment model E can be much harder. The environment is typically large and can be opaque, in that descriptions of all of its components might not be available.

As an example, consider the verification of quantitative properties, such as bounds on timing and power. Such analysis is central to the design of reliable embedded software and systems, such as those in automotive, avionics, and medical applications. However, the verification of such properties on a program is made difficult by their heavy dependence on the program's environment, which includes (at a minimum) the processor it runs on and characteristics of the memory hierarchy. Current practice requires significant manual modeling, which can be tedious, error-prone and time consuming, taking several months to create a model of a relatively simple microcontroller. It is no wonder that the state of the art in such quantitative verification problems lags far behind that in more traditional "Boolean" verification.

Another significant challenge for verification is the incompleteness of the specification Φ. Verification is only as good as the specification one verifies. The difficulty of writing complete and consistent specifications has been noted in industrial practice (see, e.g., [7]). This challenge is only exacerbated when we move from verification to *synthesis*. In automatic synthesis, one starts with a specification Φ of the system to be synthesized, along with a model of its environment E. The goal of synthesis is to generate a system S that satisfies Φ when composed with E. Figure 2 depicts the synthesis process. Modeled thus, the essence of

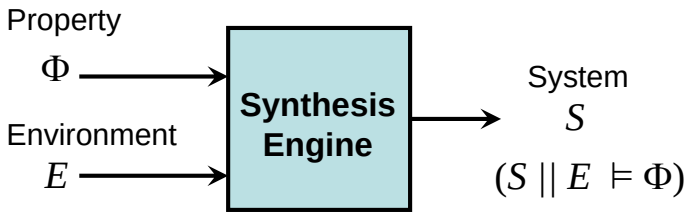

Figure 2: Formal synthesis procedure.

synthesis can be viewed as a *game solving* problem, where S and E represent the two players in a game; S is computed as a winning strategy ensuring that the composed system $S\|E$ satisfies Φ for all input sequences generated by the environment E. If such an S exists, we say that the specification (Φ and E) is *realizable*. Starting with the seminal work on automata-theoretic and deductive synthesis from specifications (e.g. [29, 38]), there has been steady progress on automatic synthesis. In particular, many recent techniques (e.g. [52, 54]) build upon the progress in formal verification in order to perform synthesis. However, there is a long way to go before automated synthesis is practical and widely applicable. One major challenge is the difficulty of

obtaining complete, formal specifications from the user. Even expert users find it difficult to write complete, formal specifications that are realizable. Often, when they do write complete specifications, the effort to write these is arguably more than that required to manually create the design in the first place. In some cases, one can write the specification as a easier-to-write but unoptimal system description (e.g., [52]). However, in many situations, the specification writing problem serves as a significant hindrance for automatic synthesis.

Additionally, the challenge of modeling the environment, as discussed above for verification, also remains for synthesis. Finally, synthesis problems typically have greater computational complexity[1] than verification problems for the same class of specifications and models. For instance, equivalence checking of combinational circuits is NP-complete and routinely solved in industrial practice, whereas synthesizing a combinational circuit from a finite set of components is Σ_2-complete and only possible in limited settings in practice. In some cases, both verification and synthesis are undecidable, but there are still compelling reasons to have efficient procedures in practice; a good example is controller synthesis for hybrid systems — systems with both discrete and continuous state — whose continuous dynamics is non-linear.

Although verification and synthesis are usually studied as separate problems, they are closely related. Verification problems, such as model checking finite-state systems, often involve synthesis tasks, such as the task of synthesizing inductive invariants, environment assumptions, or abstractions. Similarly, synthesizers employ verifiers, not the least to verify that the systems they generate are correct. Synthesis tools use many of the same computational engines that verification tools do. In fact, many recent synthesis techniques are essentially built on top of verifiers for the same class of systems (e.g., [52, 54]). Given this tight connection, it is no surprise that automatic verification and synthesis share many of the same challenges, as we have outlined above. To recapitulate, these challenges are:

- Difficulty of generating precise environment models;
- Lack of good formal specifications, and
- High computational complexity.

Some of these challenges — such as dealing with computational complexity — can be partially addressed by advances in *computational engines* such as Binary Decision Diagrams (BDDs) [9], Boolean satisfiability (SAT) [28], and satisfiability modulo theories (SMT) solvers [6]. However, these alone are not sufficient to extend the reach of formal methods for verification and synthesis. New *methodologies* are also required.

In this position paper, we present one such methodology that, in our experience so far, appears promising. The central idea of this approach, which we term *sciduction*,[2] is to tightly integrate *induction*, *deduction*, and *structure hypotheses*. *Induction* is the process of inferring a general law or principle from observation of particular instances. Machine learning algorithms are typically inductive, generalizing from (labeled) examples to obtain the learned concept or classifier [34, 3]. *Deduction*, on the other hand, involves the use of general rules and axioms to infer conclusions about particular problem instances. Traditional formal verification and synthesis techniques, such as model checking or theorem proving, are deductive. This is no surprise, as formal verification and synthesis problems (see Figures 1 and 2) are, by their very nature, deductive processes: given a particular specification Φ, environment E, and system S, a verifier typically uses a rule-based decision procedure for that class of Φ, E, and S to deduce if $S \| E \models \Phi$. On the other hand, inductive reasoning may seem out of place here, since typically an inductive argument only ensures that the truth of its premises make it *likely or probable* that its conclusion is also true. However, we argue in this paper that one can combine inductive and deductive reasoning to obtain the kinds of guarantees one desires in formal verification and synthesis. The key is the use of *structure hypotheses*. These are mathematical hypotheses used to define the class

[1] Somewhat abusing terminology, in this paper we will use "computational complexity" to refer to problem hardness both in theoretical and practical respects.

[2] sciduction stands for structure-constrained induction and deduction. It is a term coined to simply put a descriptive label on the approach for ease of future reference.

of artifacts to be synthesized within the overall verification or synthesis problem. The sciductive approach constrains inductive and deductive reasoning using structure hypotheses, and actively combines inductive and deductive reasoning: for instance, deductive techniques generate examples for learning, and inductive reasoning is used to guide the deductive engines. We describe the methodology in detail, with comparison to related work, in Section 2.

Let us reflect a bit on the combined use of induction and deduction for verification and synthesis. Automatic techniques for verification and synthesis are typically deductive. However, one can argue that humans often employ a combination of inductive and deductive reasoning while performing verification or synthesis. For example, while proving a theorem, one often starts by working out examples and trying to find a pattern in the properties satisfied by those examples. The latter step is a process of inductive generalization. These patterns might take the form of lemmas or background facts that then guide a deductive process of proving the statement of the theorem from known facts and previously established theorems (rules). Similarly, while creating a new design, one often starts by enumerating sample behaviors that the design must satisfy and hypothesizing components that might be useful in the design process; one then systematically combines these components, using design rules, to obtain a candidate artifact. The process usually iterates between inductive and deductive reasoning until the final artifact is obtained. It is this combination of inductive and deductive reasoning that we seek to formalize using the notion of sciduction.

We demonstrate the sciductive approach to automatic verification and synthesis using three applications:

1. Timing analysis of software (Section 3);
2. Synthesis of loop-free programs (Section 4), and
3. Synthesizing switching logic for control of hybrid systems (Section 5).

The first application of the sciduction approach addresses the problem of environment modeling; the second, the problem of incomplete specifications, and the third tackles the problem of high computational complexity. Some future applications are also explored in Section 6.

2 Sciduction: Formalization and Related Work

We begin with a formalization of the sciductive approach, and then compare it to related work. This section assumes some familiarity with basic terminology in formal verification and machine learning — see the relevant books by Clarke et al. [13], Manna and Pnueli [30], and Mitchell [34] for an introduction.

2.1 Verification and Synthesis Problems

As discussed in Section 1, an instance of a verification problem is defined by a triple $\langle S, E, \Phi \rangle$, where S denotes the system, E is the environment, and Φ is the property to be verified. Here we assume that S, E, and Φ are described formally, in mathematical notation. Similarly, an instance of a synthesis problem is defined by the pair $\langle E, \Phi \rangle$, where the symbols have the same meaning. In both cases, as noted earlier, it is possible *in practice* for the descriptions of S, E, or Φ to be missing or incomplete; in such cases, the missing components must be synthesized as part of the overall verification or synthesis process.

A *family of verification or synthesis problems* is a triple $\langle \mathcal{C}_S, \mathcal{C}_E, \mathcal{C}_\Phi \rangle$ where $\mathcal{C}_S$ is a formal description of a class of systems, $\mathcal{C}_E$ is a formal description of a class of environment models, and $\mathcal{C}_\Phi$ is a formal description of a class of specifications. In the case of synthesis, $\mathcal{C}_S$ defines the class of systems to be synthesized.

2.2 Elements of Sciduction

An instance of sciduction can be described using a triple $\langle \mathcal{H}, I, \mathcal{D} \rangle$, where the three elements are as follows:

1. *A structure hypothesis*, $\mathcal{H}$, encodes our hypothesis about the form of the *artifact to be synthesized*, whether it be an environment model, an inductive invariant, a program, or a control algorithm (or any portion thereof);

2. *An inductive inference engine*, I, is an algorithm for *learning from examples* an artifact h defined by $\mathcal{H}$, and

3. A *deductive engine*, $\mathcal{D}$, is a *lightweight decision procedure* that applies deductive reasoning to answer queries generated in the synthesis or verification process.

We elaborate on these elements below. The context of synthesis is used to explain the central ideas in the sciductive approach. The application of these ideas to verification is symmetric.

2.2.1 Structure Hypothesis

The structure hypothesis, $\mathcal{H}$, encodes our hypothesis about the form of the artifact to be synthesized.

Formally $\mathcal{H}$ is a (possibly infinite) set of *artifacts*. $\mathcal{H}$ encodes a hypothesis that the system to be synthesized falls in a subclass $\mathcal{C}_\mathcal{H}$ of $\mathcal{C}_S$ (i.e., $\mathcal{C}_\mathcal{H} \subseteq \mathcal{C}_S$). Note that $\mathcal{H}$ needs not be the same as $\mathcal{C}_\mathcal{H}$, since the artifact being synthesized might just be a portion of the full system description, such as the guard on transitions of a state machine, or the assignments to certain variables in a program. Each artifact $h \in \mathcal{H}$, in turn, corresponds to a unique set of *primitive elements* that defines its semantics. The form of the primitive element depends on the artifact to be synthesized.

More concretely, here are two examples of a structure hypothesis $\mathcal{H}$:

1. Suppose that $\mathcal{C}_S$ is the set of all finite automata over a set of input variables V and output variables U satisfying a specification Φ. Consider the structure hypothesis $\mathcal{H}$ that restricts the finite automata to be synchronous compositions of automata from a finite library L. The artifact to be synthesized is the entire finite automaton, and so, in this case, $\mathcal{H} = \mathcal{C}_\mathcal{H}$. Each element $h \in \mathcal{H}$ is one such composition of automata from L. A primitive element is an input-output trace in the language of the finite automaton h.
2. Suppose that $\mathcal{C}_S$ is the set of all hybrid automata [1], where the guards on transitions between modes can be any region in $\mathbb{R}^n$ but where the modes of the automaton are fixed. A structure hypothesis $\mathcal{H}$ can restrict the guards to be hyperboxes in $\mathbb{R}^n$ — i.e., conjunctions of upper and lower bounds on continuous state variables. Each $h \in \mathcal{H}$ is one such hyperbox, and a primitive element is a point in h. Notice that $\mathcal{H}$ defines a subclass of hybrid automata $\mathcal{C}_\mathcal{H} \subset \mathcal{C}_S$ where the guards are n-dimensional hyperboxes. Note also that $\mathcal{H} \neq \mathcal{C}_\mathcal{H}$ in this case.

A structure hypothesis $\mathcal{H}$ can be syntactically described in several ways. For instance, in the second example above, $\mathcal{H}$ can define a guard either set-theoretically as a hyperbox in $\mathbb{R}^n$ or using mathematical logic as a conjunction of atomic formulas, each of which is an interval constraint over a real-valued variable.

2.2.2 Inductive Inference

The inductive inference procedure, I, is an algorithm for learning from examples an artifact $h \in \mathcal{H}$.

While any inductive inference engine can be used, in the context of verification and synthesis we expect that the learning algorithms I have one or more of the following characteristics:

- I performs *active learning*, selecting the examples that it learns from.

- Examples and/or labels for examples are generated by one or more *oracles*. The oracles could be implemented using deductive procedures or by evaluation/execution of a model on a concrete input. In some cases, an oracle could be a human user.
- A deductive procedure is invoked in order to synthesize a concept (artifact) that is consistent with a set of labeled examples. The idea is to formulate this synthesis problem as a decision problem where the concept to be output is generated from the satisfying assignment.

2.2.3 Deductive Reasoning

The deductive engine, $\mathcal{D}$, is a lightweight decision procedure that applies deductive reasoning to answer queries generated in the synthesis or verification process.

The word "lightweight" refers to the fact that the decision problem being solved by $\mathcal{D}$ must be easier, in theoretical or practical terms, than that corresponding to the overall synthesis or verification problem.

In theoretical terms, "lightweight" means that at least one of the following conditions must hold:

1. $\mathcal{D}$ must solve a problem that is a strict special case of the original.
2. One of the following two cases must hold:
   (i) If the original (synthesis or verification) problem is decidable, and the worst-case running time of the best known procedure for the original problem is $O(T(n))$, then $\mathcal{D}$ must run in time $o(T(n))$.
   (ii) If the original (synthesis or verification) problem is undecidable, $\mathcal{D}$ must solve a decidable problem.

In practical terms, the notion of "lightweight" is fuzzier: intuitively, the class of problems addressed by $\mathcal{D}$ must be "more easily solved in practice" than the original problem class. For example, $\mathcal{D}$ could be a finite-state model checker that is invoked only on abstractions of the original system produced by, say, localization abstraction [24] — it is lightweight if the abstractions are solved faster in practice than the original concrete instance. Due to this fuzziness, it is preferable to define "lightweight" in theoretical terms whenever possible.

$\mathcal{D}$ can be used to answer queries generated by I, where the query is typically formulated as a decision problem to be solved by $\mathcal{D}$. Here are some examples of tasks $\mathcal{D}$ can perform and the corresponding decision problems:

- Generating examples for the learning algorithm.
  Decision problem: "does there exist an example satisfying the criterion of the learning algorithm?"
- Generating labels for examples selected by the learning algorithm.
  Decision problem: "is L the label of this example?"
- Synthesizing candidate artifacts.
  Decision problem: "does there exists an artifact consistent with the observed behaviors/examples?"

2.2.4 Discussion

We now make a few remarks on the formalization of sciduction introduced above.

In the above description of the structure hypothesis, $\mathcal{H}$ only "loosely" restricts the class of systems to be synthesized, allowing the possibility that $\mathcal{C}_\mathcal{H} = \mathcal{C}_S$. We argue that a tighter restriction is desirable. One important role of the structure hypothesis is to reduce the search space for synthesis, by restricting the class of artifacts $\mathcal{C}_S$. For example, a structure hypothesis could be a way of codifying the form of human insight to be provided to the synthesis process. Additionally, restricting $\mathcal{C}_\mathcal{H}$ also aids in inductive inference. Fundamentally, the effectiveness of inductive inference (i.e., of I) is limited by the examples presented to it as input; therefore, it is important not only to select examples carefully, but also for the inference to generalize well beyond the presented examples. For this purpose, the structure hypothesis should place a strict restriction on the search space, by which we mean that $\mathcal{C}_\mathcal{H} \subsetneq \mathcal{C}_S$. The justification for this stricter restriction comes from the importance of *inductive bias* in machine learning. Inductive bias is the set of assumptions required to

deductively infer a concept from the inputs to the learning algorithm [34]. If one places no restriction on the type of systems to be synthesized, the inductive inference engine I is unbiased; however, an unbiased learner will learn an artifact that is consistent only with the provided examples, with no generalization to unseen examples. As Mitchell [34] writes: "a learner that makes no a priori assumptions regarding the identity of the target concept has no rational basis for classifying any unseen instances." Given all these reasons, it is highly desirable for the structure hypothesis $\mathcal{H}$ to be such that $\mathcal{C}_{\mathcal{H}} \subsetneq \mathcal{C}_{\mathcal{S}}$. This is the case in all the applications we demonstrate herein.

Another point to note is that it is possible to use randomization in implementing I and $\mathcal{D}$. For example, a deductive decision procedure that uses randomization can generate a YES/NO answer with high probability.

Next, although we have defined sciduction as combining a single inductive engine with a single deductive engine, this is only for simplicity of the definition and poses no fundamental restriction. One can always view multiple inductive (deductive) engines as a being contained in a single inductive (deductive) procedure where this outer procedure passes its input to the appropriate "sub-engine" based on the type of input query.

Finally, in our definition of sciduction, we do not advocate any particular technique of combining inductive and deductive reasoning. Indeed, we envisage that there are many ways to "configure" the combination of $\mathcal{H}$, $\mathcal{D}$, and I, perhaps using inductive procedures within deductive engines and vice-versa. Any mode of integrating $\mathcal{H}$, I, and $\mathcal{D}$ that satisfies the requirements stated above on each of those three elements is admissible. We expect that the particular requirements of each application will define the mode of integration that works best for that application. We present illustrative examples in Sections 3, 4, and 5.

In Section 2.3, we discuss several examples of related work, to differentiate the sciductive approach from other methods.

2.2.5 Soundness and Completeness Guarantees

It is highly desirable for verification or synthesis procedures to provide *soundness* and *completeness* guarantees.

A verifier is said to be *sound* if, given an arbitrary problem instance $\langle S, E, \Phi \rangle$, the verifier outputs "YES" only if $S \| E \models \Phi$. The verifier is said to be *complete* if it outputs "NO" when $S \| E \not\models \Phi$.

The definitions for synthesis are similar. A synthesis technique is *sound* if, given an arbitrary problem instance $\langle E, \Phi \rangle$, if it outputs S, then $S \| E \models \Phi$. A synthesis technique is *complete* if, when there exists S such that $S \| E \models \Phi$, it outputs at least one such S.

Note that we can have probabilistic analogs of soundness and completeness. Informally, a verifier is *probabilistically sound* if it is sound with "high probability;" we will leave a more precise discussion of this point to a later stage in this paper when it becomes relevant.

In sciduction, whether we get soundness and completeness guarantees depends on the validity of the structure hypothesis. We say that the structure hypothesis $\mathcal{H}$ is *valid* if the artifact to be synthesized, if one exists, is guaranteed to be an element of the class $\mathcal{C}_{\mathcal{H}}$.

Given the above definition, the main requirement we impose on sciduction is as follows:

> Every sciductive procedure, whether for verification or synthesis, must be *sound* (or probabilistically sound) *if the structure hypothesis is valid*.

Without such a requirement, the sciductive procedure is a heuristic, best-effort verification or synthesis procedure. (It could be extremely useful, nonetheless.) With this requirement, we have a mechanism to formalize the assumptions under which we obtain soundness — namely, the structure hypothesis.

While completeness is also desirable, we will not require this of a sciductive procedure, since for many problems this can be impossible to guarantee (e.g., due to undecidability).

2.3 Comparison with Related Work

In both ancient and modern philosophy, there is a long history of arguments about the distinction between induction and deduction and their relationship and relative importance. This literature, although very interesting, is not directly relevant to the discussion in this paper.

Within computer science and engineering, the field of artificial intelligence (AI) has long studied inductive and deductive reasoning and their connections (see, e.g., [45]). As mentioned earlier, Mitchell [34] describes how inductive inference can be formulated as a deduction problem where inductive bias is provided as an additional input to the deductive engine. *Inductive logic programming* [36], an approach to machine learning, blends induction and deduction by performing inference in first-order theories using examples and background knowledge. Combinations of inductive and deductive reasoning have also been explored for synthesizing programs (plans) in AI; for example, the SSGP approach [16] generates plans by sampling examples, generalizing from those samples, and then proving correctness of the generalization.

Our focus is on the use of combined inductive and deductive reasoning in *formal verification and synthesis*. While several techniques for verification and synthesis combine subsets of induction, deduction, and structure hypotheses, there are important distinctions between many of these and the sciduction approach. We highlight these distinctions below using a representative sample of related work.

2.3.1 Closely Related Work

We first survey prior work in verification and synthesis that has provided inspiration for formulating the sciductive approach. We note that many of these prior techniques can be formulated as instances of sciduction. Indeed, sciduction can be seen as a "lens" through which we view the key ideas amongst these techniques so as to extend and apply them to new problem domains.

Counterexample-Guided Abstraction Refinement (CEGAR). In CEGAR [12], an abstract model is synthesized so as to eliminate spurious counterexamples. One can view CEGAR as an instance of sciduction. The form of the abstraction function — for example, localization abstraction [24] — is a structure hypothesis. The deductive engine $\mathcal{D}$, for finite-state model checking, is the model checker. The inductive engine I is an algorithm to learn from counterexamples, which can be viewed as *equivalence queries* in the Angluin model [2]. The use of alternative learning algorithms to generate abstractions is also possible, as demonstrated by Gupta [18].

Learning for Assume-Guarantee Reasoning and Compositional Verification. The use of learning algorithms has been investigated extensively in the context of synthesizing environment assumptions for compositional verification. Most of these techniques are based on Angluin's L^* algorithm and its variants; see [15] for a recent collection of papers on this topic. These techniques are almost an instance of sciduction, similar to CEGAR, but they typically make no restrictive structure hypothesis (i.e., $\mathcal{C}_\mathcal{H} = \mathcal{C}_S$). For example, for techniques that target finite-state model checking, the synthesized environment assumptions can be any finite-state machine that interfaces with the system. One possible restriction would be to limit the input or output alphabets for the learning algorithm.

Software Synthesis by Sketching. Programming by sketching is a novel approach to synthesizing software by encoding programmer insight in the form of a *partial program*, or "sketch" [51, 52, 50]. The algorithmic component of this approach can be viewed as an instance of sciduction. The partial program imposes a structure hypothesis on the synthesis problem. An algorithmic approach central to this work is counterexample-guided inductive synthesis (CEGIS) [52, 50], which operates in a manner similar to CEGAR. Like CEGAR,

the inductive engine I is based on Angluin's query-based learning model, and the learning is based on counterexamples generated by a verifier ($\mathcal{D}$) — typically based on a satisfiability solver or model checker. Since CEGIS repeatedly calls a verifier in its main loop, it is effective for systems where the verification problem is significantly easier than the synthesis problem for the same class of systems and specifications. For example, in the synthesis of finite (bit-vector) loop-free programs [52], the verifier solves SAT, whereas the synthesis problem is expressible as a 2-QBF problem (satisfiability of quantified Boolean formulas with one alternation).

Discussion. Techniques such as CEGAR, CEGIS, and learning-based compositional verification have proved very successful and influential in their respective domains. However, the reader may note that the problems addressed in Sections 3, 4, and 5 are all problems that are difficult to tackle using techniques such as CEGIS or CEGAR for the reasons described in Section 1 — the lack of a precise specification or environment model, or the high computational complexity of verification that makes it difficult to repeatedly invoke a verifier within a synthesis loop. Our goal at formalizing the notion of sciduction is to provide a common framework to build upon these successes and go beyond the problems addressed by these methods.

2.3.2 Other Related Work

We now highlight some other related work and their distinctions with sciduction.

Verification-Driven Synthesis. Srivastava et al. [54] have proposed a verification-driven approach to synthesis (called VS3), where programs with loops can be synthesized from a scaffold comprising of a logical specification of program functionality, and domain constraints and templates restricting the space of synthesizable programs. The latter is a structure hypothesis. However, the approach is not sciduction since the synthesis techniques employed are purely deductive in nature. More recently, Srivastava et al. [53] have proposed a path-based inductive synthesis approach. An inverse of a program is synthesized by exploring a small subset of that program's paths — the inductive generalization is that the program synthesized to yield the inverse for inputs corresponding to that small path subset is also deemed to yield the correct inverse for all other paths. In contrast with sciduction, there is no guarantee that under the structure hypothesis, the synthesis routine will yield the correct program, if one exists.

Boolean Satisfiability (SAT) Solving. Modern SAT solvers use the *conflict-driven clause learning* (CDCL) approach [27]. In a CDCL SAT solver, whenever the solver explores a partial assignment that leads to a conflict (i.e., the formula evaluates to false), it "learns" a new clause that captures a reason for that conflict. Thus, one can view CDCL SAT solving as using a subroutine (clause learning) that generalizes from experiments, each experiment being a partial assignment explored by the SAT solver. However, such an approach is distinct from sciduction. The clause learning is not inductive; it is in fact a *deductive procedure*, essentially being a way of applying the resolution proof rule to the clauses involved in implications generated by the falsifying assignment.

Program Analysis Using Relevance Heuristics. McMillan [33] describes the idea of verification based on "relevance heuristics", which is the notion that facts useful in proving special cases of the verification problem are likely to be useful in general. This idea is motivated by the similar approach taken in (CDCL) SAT solvers. A concrete instance of this approach is interpolation-based model checking [32], where a proof of a special case (e.g., the lack of an assertion failure down a certain program path) is used to generate facts relevant to solving the general verification problem (e.g., correctness along all program paths). Although this work generalizes from special cases, the generalization is not inductive, and no structure hypothesis is involved.

Automata-Theoretic Synthesis from Linear Temporal Logic (LTL). One of the classic approaches to synthesis is the automata-theoretic approach for synthesizing a finite-state transducer (FST) from an LTL specification, pioneered by Pnueli and Rosner [38]. The approach is a purely deductive one, with a final step that involves solving an emptiness problem for tree automata. No structure hypothesis is made on the FST being synthesized. Although advances have been made in the area of synthesis from LTL, for example in special cases [37], some major challenges remain: (i) writing complete specifications is tedious and error-prone, and (ii) the computational complexity for general LTL is doubly-exponential in the size of the specification. It would be interesting to explore if inductive techniques can be combined with existing deductive automata-theoretic procedures to form an effective sciductive approach to some class of systems or specifications.

3 Quantitative Analysis of Sofware

The analysis of quantitative properties, such as bounds on timing and power, is central to the design of reliable embedded software and systems, such as those in automotive, avionics, and medical applications. Fundamentally, such properties depend not only on program logic, but also on details of the program's environment. The environment includes many things — the processor, characteristics of the memory hierarchy, the operating system, the network, etc. Moreover, in contrast with many other verification problems, the environment must be modeled with a relatively high degree of precision. Most state-of-the-art approaches to worst-case execution time (WCET) analysis require significant manual modeling, which can be tedious, error-prone and time consuming, even taking several months to create a model of a relatively simple microcontroller. See [46] for a more detailed description of the challenges in quantitative analysis of software.

3.1 The Problem

For simplicity, we will consider the following representative timing analysis problem:

⟨TA⟩ Given a terminating program P, its platform (environment) E, and a fixed starting state of E, is the execution time of P on E always at most τ?

If the execution time can exceed τ, it is desirable to obtain a test case (comprising a state of P) that shows how the bound of τ is exceeded.

The main challenge in solving this problem, as noted earlier, is the generation of a model of the environment E. We illustrate this challenge briefly using a toy example. The complexity of the timing analysis arises from two dimensions of the problem: the *path dimension*, where one must find the right computation path in the task, and the *state dimension*, where one must find the right (starting) environment state to run the task from. Moreover, these two dimensions interact closely; for example, the choice of path can affect the impact of the starting environment state.

Consider the toy C program in Fig. 3(a). It contains a loop, which executes at most once. Thus, the control-flow graph (CFG) of the program can be unrolled into a directed acyclic graph (DAG), as shown in Fig. 3(b). Suppose we execute this program on a simple processor with an in-order pipeline and a data cache. Consider executing this program from the state where the cache is empty. The final statement of the program, `*x += 2`, contains a load, a store, and an arithmetic operation. If the left-hand path is taken, the load will suffer a cache miss; however, if the right-hand path is taken, there is a cache hit. The difference in timing between a cache hit and a miss can be an order of magnitude. Thus, the time taken by this statement depends on the program path taken. However, if the program were executed from a state with the data in the cache, there will be a cache hit even if the left-hand path is taken.

Thus, even with this toy program and a simple processor, one can observe that a timing analysis tool must explore the space of all possible program paths and all possible environment states – both potentially

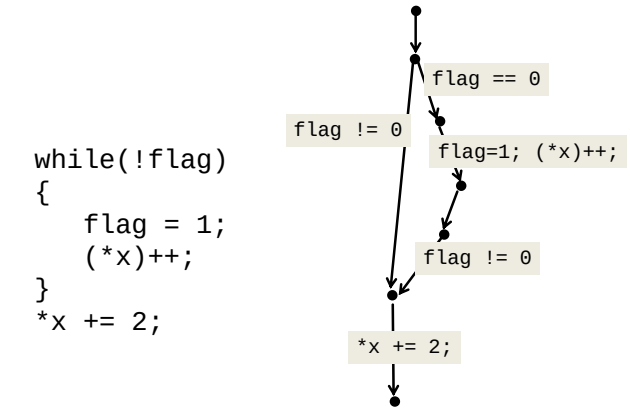

Figure 3: **Simple Illustrative Example**

exponentially-large search spaces. Most WCET tools assume a known starting environment state, and attempt to predict worst-case timing by inferring worst-case environment states at basic block boundaries. For this, they must (i) model the platform precisely to predict timing at basic block boundaries, and (ii) search an exponentially-large environment state space at these boundaries. If the starting environment state is unknown, the problem is even harder!

We show in the next section that sciduction offers a promising approach to address this challenge of environment modeling.

3.2 Our Approach: GAMETIME

Automatic inductive inference of models offers a way to mitigate the challenge of environment modeling. In our approach, termed GAMETIME, a *program-specific timing model* of the platform is inferred from observations of the program's timing that are automatically and systematically generated. The program-specificity is an important difference from traditional approaches, which seek to manually construct a timing model that works for *all* programs one might run on the platform. GAMETIME only requires one to run end-to-end measurements on the target platform, making it easy to port to new platforms.

The central idea in GAMETIME is to view the platform as an adversary that controls the choice and evolution of the environment state, while the tool has control of the program path space. The problem is then a formulated as a game between the tool and the platform. GAMETIME uses a sciductive approach to solve this game based on the following elements:

- *Structure hypothesis:* The platform E is modeled as an adversarial process that selects weights on the edges of the control-flow graph of the program P in two steps: first, it selects the path-independent weights w, and then the path-dependent component π. Formally, w cannot depend on the program path being executed, whereas π is drawn from a distribution which is a function of that path. Both w and π are elements of $\mathbb{R}^m$, where m is the number of edges in the CFG after unrolling loops and inlining function calls. We term w as the *weight* and π as the *perturbation*, and the structure hypothesis as the *weight-perturbation model*.
  To summarize, each artifact in the structure hypothesis $\mathcal{H}$ is a process that selects a pair (w, π) every time the program P runs, where additionally the pair must satisfy the following constraints:
  1. The mean perturbation along any path is bounded by a quantity $\mu_{\max}$.
  2. (for worst-case analysis) The worst-case path is the unique longest path by a margin.
- *Inductive inference:* The inductive inference routine is a learning algorithm that operates in a *game-*

theoretic online setting. The task of this algorithm is to learn the (w,π) model from measurements. The idea in GAMETIME is to measure execution times of P along so-called *basis paths*, choosing amongst these basis paths uniformly at random over a number of trials. A (w,π) model is inferred from the end-to-end measurements of program timing along each of the basis paths. See [49] for details.

- *Deductive reasoning:* An SMT solver forms the deductive engine for GAMETIME. The basis paths constitute the examples for the inductive learning algorithm. These examples are generated using SMT-based test generation — from each candidate basis path, an SMT formula is generated such that the formula is satisfiable iff the path is feasible.

The GAMETIME approach, along with an exposition of theoretical and experimental results, including comparisons with other methods, is described in existing papers [48, 49, 47]. We only give a brief overview here.

Figure 4 depicts the operation of GAMETIME. As shown in the top-left corner, the process begins with the generation of the control-flow graph (CFG) corresponding to the program, where all loops have been unrolled to a maximum iteration bound, and all function calls have been inlined into the top-level function. The CFG is assumed to have a single source node (entry point) and a single sink node (exit point); if not, dummy source and sink nodes are added. The next step is a critical one, where a subset of program paths, called *basis paths* are extracted. These basis paths are those that form a basis for the set of all paths, in the standard linear algebra sense of a basis. A *satisfiability modulo theories (SMT) solver* — the deductive engine — is invoked to ensure that the generated basis paths are feasible. For each feasible basis path generated, the SMT solver generates a test case that drives program execution down that path. Thus a set of *feasible basis paths* is generated that spans the entire space of feasible program paths, along with the corresponding test cases.

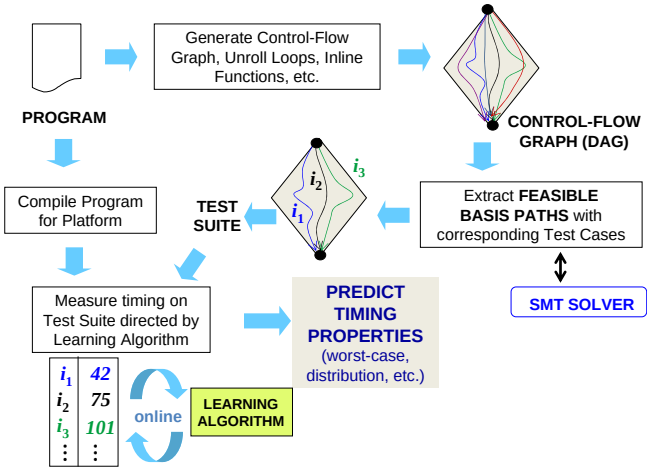

Figure 4: GAMETIME **overview**

The program is then compiled for the target platform, and executed on these test cases. In the basic GAMETIME algorithm (described in [48, 49]), the sequence of tests is randomized, with basis paths being chosen uniformly at random to be executed. The overall execution time of the program is recorded for each test case. From these end-to-end execution time measurements, GAMETIME's learning algorithm generates the (w,π) model that can then be used for timing analysis. In contrast with most existing tools for timing analysis (see, e.g., [42]), GAMETIME can not only be used for WCET estimation, it can also be used to predict execution time of arbitrary program paths, and certain execution time statistics (e.g., the distribution of times). For example, to answer the problem $\langle \text{TA} \rangle$ presented in the preceding section, GAMETIME would predict the longest path, execute it to compute the corresponding timing τ^*, and compare that time with τ: if $\tau^* \leq \tau$, then GAMETIME returns "YES", otherwise it returns "NO" along with the corresponding test case.

3.3 Guarantees and Results

Assuming the structure hypothesis holds, GAMETIME answers the timing analysis question $\langle \text{TA} \rangle$ with high probability. In other words, if the structure hypothesis is valid, GAMETIME is *probabilistically sound and complete* in the following sense:

> Given any $\delta > 0$, if one runs a number of tests that is polynomial in $\ln \frac{1}{\delta}$, $\mu_{\max}$, and the program parameters, GAMETIME will report the correct YES/NO answer to Problem $\langle \text{TA} \rangle$ with probability at least $1 - \delta$.

See the theorems in [48, 49] for details.

Experimental results indicate that in practice GAMETIME can accurately predict not only the worst-case path (and thus the WCET) but also the distribution of execution times of a task from various starting environment states. As a sample result, we used GAMETIME to estimate the distribution of execution times of a modular exponentiation function function for an 8-bit exponent (256 program paths) by testing only (9) basis paths. The experiments were performed for the StrongARM-1100 processor – which implements the ARM instruction set with a 5-stage pipeline and both data and instruction caches – using the SimIt-ARM cycle-accurate simulator [40]. Fig. 5 shows the predicted and actual distribution of execution times – we see that

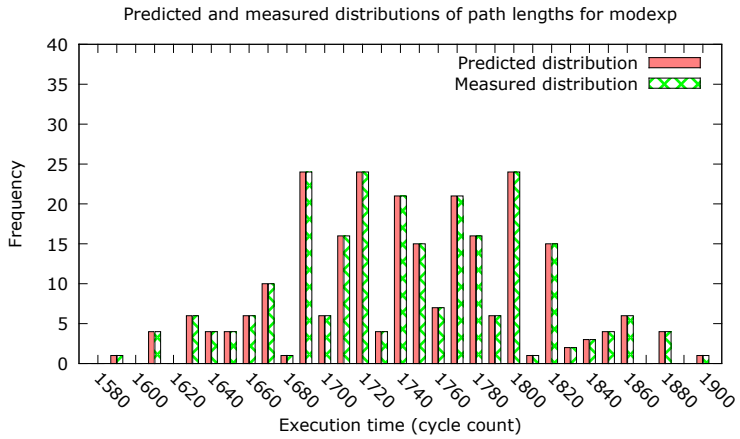

Figure 5: **Actual (striped) and Predicted (shaded) Execution Times for Modular Exponentiation Code.**

GAMETIME predicts the distribution perfectly. Also, GAMETIME correctly predicts the WCET (and produces the corresponding test case: the 8-bit exponent is 255).

4 Component-Based Program Synthesis

Automatic synthesis of programs has long been one of the holy grails of software engineering. In this section, we focus on a particular application of synthesis to *program understanding* — viz., *program deobfuscation* with a focus on malicious programs. The need for deobfuscation techniques has arisen in recent years, especially due to an increase in the amount of malicious code (malware) that tends to be obfuscated [55]. Currently, human experts use decompilers and manually deobfuscate the resulting code (see, e.g., [39]). Clearly, this is a tedious task that could benefit from automated tool support.

4.1 The Problem

Our idea is to view the malware deobfuscation problem as a program (re-)synthesis problem [19]. The focus is on re-synthesizing fragments of the original malicious program where each such fragment, though it may contain loops, is essentially equivalent to a loop-free program. The main challenge for this synthesis problem is the lack of a good formal specification — in fact, the only available "specification" is the obfuscated malicious program itself. This is not an ideal specification to work with; not only might it contain maliciously-inserted constructs that make static analysis and satisfiability solving hard, its translation to a logical specification is likely to generate a large, complex formula.

Our approach to this problem is to view the obfuscated program as an *I/O oracle* that maps a given program input (starting state) to the desired output (ending state).[3] The problem is then to synthesize the program using a small number of queries to the I/O oracle.

An important advantage of this I/O oracle view is that the complexity of the synthesis procedure (e.g., the number of queries to the I/O oracle) required is a function of the intrinsic functionality of the program, not of the syntactic obfuscations applied to it.

4.2 Our Approach

Our sciductive approach to this synthesis problem has the following ingredients:

- *Structure Hypothesis:* Programs are assumed to be loop-free compositions of components drawn from a finite component library L. Each component in this library implements a programming construct that is essentially a *bit-vector circuit* — the outputs are bit-vector functions of a set of input bit-vectors. Conditional statements are allowed, but loops are disallowed. Thus, $\mathcal{C}_{\mathcal{H}}$ is the set of all programs that can be synthesized as syntactically legal compositions of components from L.
- *Inductive Inference:* The inductive inference routine is an algorithm that learns a program from *distinguishing inputs* — those inputs that can distinguish between non-equivalent programs in $\mathcal{C}_{\mathcal{H}}$ which are consistent with past interaction with the I/O oracle. The inductive routine starts with one or more randomly chosen inputs and their outputs obtained from the I/O oracle. On each iteration, the routine constructs an SMT formula whose satisfying assignment yields a program consistent with all input-output examples seen so far. It also queries the SMT solver for another such program which is semantically different from the first, as well as a distinguishing input that demonstrates this semantic difference. If no such alternative program exists, the process terminates. Otherwise, the process is repeated until a semantically unique program is obtained.

  Our algorithm is motivated by the characterization of the *optimal teaching sequence* by Goldman and Kearns [17]. In that paper, the authors introduce the concept of *teaching dimension* of a concept class as the minimum number of examples a teacher (oracle) must reveal to uniquely identify *any* target concept from that class. They show that the generation of an *optimal teaching sequence* of examples is equivalent to a minimum set cover problem. In the set cover problem for a given target concept, the universe of elements is the set of all incorrect concepts (programs) and each set S_i, corresponding to example x_i, contains concepts that are differentiated from the target concept by this example x_i. Our algorithm computes such a distinguishing example in each iteration, and terminates when it has computed a "set cover" that distinguishes the target concept from all other candidate concepts (the "universe"). In practice, our algorithm has required only a small number of iterations, indicating that the deobfuscation examples we consider have small teaching dimension.
- *Deductive Reasoning:* This engine is an SMT solver that performs two functions as noted above: (i) it generates candidate programs consistent with generated input-output examples; and (ii) it generates new

[3] This view of a specification as an I/O oracle applies to many other contexts, including programming by demonstration and end-user programming, where the I/O oracle may be a human user.

inputs that distinguish between two non-equivalent programs in $\mathcal{C}_{\mathcal{H}}$ consistent with the generated input-output examples.

The above instance of sciduction, although motivated by the malware deobfuscation problem, can be applied in other program synthesis settings as well.

4.3 Theoretical Guarantees and Sample Results

The structure hypothesis is valid if the library of components defines a space of target programs $\mathcal{C}_{\mathcal{H}}$ containing one that is equivalent to the obfuscated program. If the structure hypothesis is valid, the sciductive approach sketched above and presented in [19] is *sound*. The program it generates is guaranteed to be the correct program (equivalent to the obfuscated program we start with). See Theorem 4 in [19] for details.

If, however, the structure hypothesis is invalid, then our approach could either report that the problem is unrealizable (i.e., there is no program synthesizable with the component library that matches the input-output examples) or it could output a program that is consistent with all seen input-output examples, but which is not the correct program. Figure 6 depicts the possible cases.

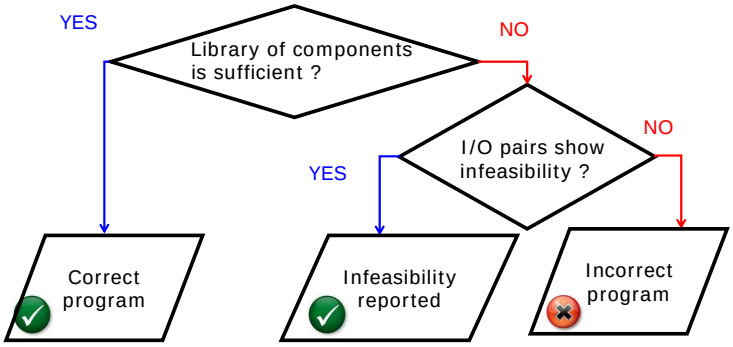

Figure 6: **Theoretical Guarantees for Program Synthesis Technique.**

In practice a range of heuristics are used to compute a sufficient component library L. Figure 7 shows two obfuscated programs and their deobfuscations computed using our approach. Both programs were deobfuscated in less than half a second.

5 Switching Logic Synthesis

Cyber-physical systems, which integrate physical dynamics and computational systems, are often conveniently modeled as multi-modal dynamical systems (MDSs). An MDS is a physical system (also known as a "plant") that can operate in different modes. The dynamics of the plant in each mode is known, and is usually specified using a continuous-time model such as a system of ordinary differential equations (ODEs). However, to achieve safe and efficient operation, it is typically necessary to switch between the different operating modes using carefully construced *switching logic*: guards on transitions between modes. The MDS along with its switching logic consitutes a *hybrid system*. Manually designing switching logic so as to ensure that the hybrid system satisfies its specification can be tricky and tedious.

While several techniques for switching logic synthesis have been proposed (see, e.g., [4, 57, 23, 31, 35, 14, 56]), it remains quite challenging to handle systems with a combination of rich discrete structure (in the form of multiple modes) and complex non-linear dynamics within modes. We discuss one such representative switching logic synthesis problem below.

P1: Interchange the source and destination addresses.

```
interchangeObs(IPaddress* src, IPadress* dest)
{ *src = *src ^ *dest;
  if (*src == *src ^ *dest)
   { *src = *src ^ *dest;
      if (*src == *src ^ *dest)
      { *dest = *src ^ *dest;
        if (*dest == *src ^ *dest)
        { *src = *dest ^ *src;
          return;
        }
        else
        { *src = *src ^ *dest;
          *dest = *src ^ *dest;
           return;
        }
      }
      else
        *src = *src ^ *dest;
   }
   *dest = *src ^ *dest;
   *src = *src ^ *dest;
   return;
}

interchange(IPaddress* src, IPadress* dest)
{
 *dest = *src ^ *dest;
 *src =  *src ^ *dest;
 *dest = *src ^ *dest;
 return;
}
```

P2: Multiply by 45

```
int multiply45Obs(int y)
{ a=1; b=0; z=1; c=0;
  while(1) {
    if (a == 0) {
      if (b == 0) {
        y=z+y; a =~a; b=~b;c=~c;
        if (~c) break;
      }
      else {
        z=z+y; a=~a; b=~b; c=~c;
        if (~c) break;
      }
   }
   else {
     if (b == 0) { z=y<<2; a=~a; }
     else {
       z=y << 3;
       a=~a; b=~b;
     }
   }
  }
  return y;
}

multiply45(int y)
{
  z = y << 2;
  y = z + y;
  z = y << 3;
  y = z + y;
  return y;
}
```

Figure 7: **Some deobfuscation benchmarks presented in [19].** For both benchmarks (a) and (b), the original obfuscated program is shown at the top and the resynthesized program generated by our system at the bottom.

5.1 The Problem

We consider the switching logic synthesis problem for *safety*. See [20] for formal definitions; an informal and more intuitive presentation is made here. A safety property can be viewed as a subset of evaluations to the n continuous state variables (i.e., a subset of $\mathbb{R}^n$); each evaluation is a *safe state*. A hybrid system is safe from a set of initial states if every reachable state is a safe state.

The problem of interest is as follows:

> Given a safety property, a multimodal dynamical system (MDS), and a set of initial states, synthesize switching logic for the MDS so that the resulting hybrid system is safe.

We impose no constraints on the intra-mode continuous dynamics in the MDS, other than it be deterministic and locally Lipschitz at all points [20].

In this article, we model hybrid systems using the *hybrid automaton* formalism [1]. An example switching logic synthesis problem is the 3-gear automatic transmission system depicted in Figure 8 [26]. This example

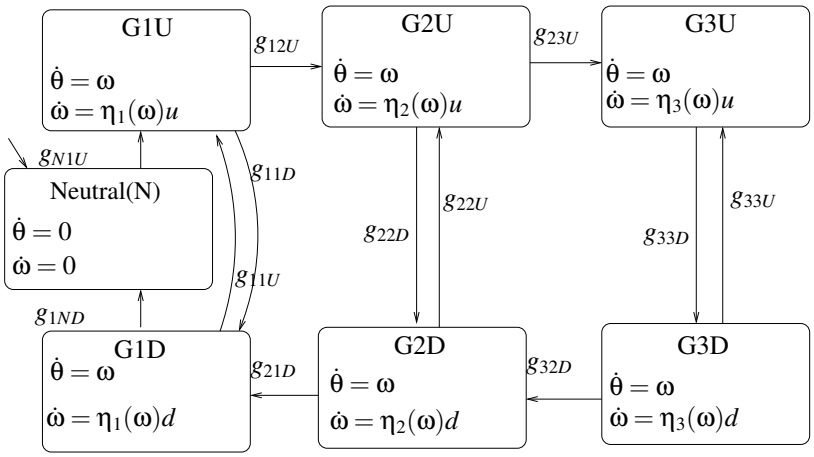

Figure 8: Automatic Transmission System

has seven modes. The transitions between modes are labeled with guard variables: g_{ij} labels the transition from Mode i to Mode j. Such a guard is termed an *entry guard* for Mode j and an *exit guard* for Mode i.

Note that for this example, the dynamics in each mode are *non-linear differential equations*. u and d denote the throttle in accelerating and deaccelerating mode. The transmission efficiency η is η_i when the system is in the ith gear, given by:

$$\eta_i = 0.99e^{-(\omega - a_i)^2/64} + 0.01$$

where $a_1 = 10, a_2 = 20, a_3 = 30$ and ω is the speed. The distance covered is denoted by θ. The acceleration in mode i is given by the product of the throttle and transmission efficiency.

For simplicity, suppose that $u = 1$ and $d = -1$. Also, let the initial state be $\theta = 0, \omega = 0$. Suppose that the system must be designed to reach $\theta = \theta_{max} = 1700$ with $\omega = 0$. The synthesis problem is to find the guards between the modes such that the efficiency η is high for speeds greater than some threshold, that is, $\omega \geq 5 \Rightarrow \eta \geq 0.5$. Also, ω must be less than an upper limit of 60. So, the safety property ϕ_S to be enforced would be

$$(\omega \geq 5 \Rightarrow \eta \geq 0.5) \wedge (0 \leq \omega \leq 60)$$

We will additionally assume that we know an initial overapproximation of the guards. Since the speed must reduce to 0 on reaching θ_{max}, the guard g_{1ND} is initialized to $\phi_S \wedge \theta = \theta_{max} \wedge \omega = 0$. All the other guards are initialized to $0 \leq \omega \leq 60$. Clearly every switching state must be a safe state.

Note that for the class of hybrid automata with nonlinear dynamics within modes, even reachability analysis is undecidable. Synthesizing safe switching logic is therefore undecidable too, unless additional assumptions are imposed. While we cannot expect to have a synthesis procedure that works in all cases, our experience is that it is possible to develop an approach that handles many cases arising in practice.

5.2 Our Approach

We adopt once again a sciductive approach to the controller synthesis problem, with the following elements:

- *Structure Hypothesis:* The essence of the structure hypothesis is to impose a particular syntactic form on the guards of the hybrid system: that the guards are hyperboxes. More precisely, the structure hypothesis includes the following two properties:

1. If there is a safe switching logic, then there exists one in which all guards are n-dimensional hyperboxes with vertices lying on a known discrete grid.[4]
2. For each mode, if all exit guards and all but one entry guard are fixed as hyperboxes, then for the remaining entry transition to that mode, the safe switching states constitute a hyperbox.

Note that the second item above could be guaranteed by constraints on the type of continuous dynamics within a mode, for example, if state variables vary monotonically within a mode [20]. Also, since the set of initial states is also a particular kind of guard (on the "transition" that initializes the hybrid system), the structure hypothesis will also apply to the set of initial states.

Given the above structure hypothesis, $\mathcal{C}_{\mathcal{H}}$ is the set of all hybrid automata in which the guards satisfy the above hyperbox restriction.

- *Inductive Inference:* This routine is an algorithm to learn hyperboxes in $\mathbb{R}^n$ from labeled examples. An example is a point in $\mathbb{R}^n$. Its label is positive if the point is inside the box, and negative otherwise.

  More specific to our problem context, the learning problem is as follows. We are given a mode with its associated entry and exit guards. These guards are assumed to be overapproximate hyperboxes — the guards of a safe switching logic, if one exists, are subsets of the corresponding overapproximate guards. Given an entry guard, which could contain both safe and unsafe switching states, we want to infer a hyperbox that contains only the safe switching states and none of the unsafe switching states.

  If the structure hypothesis is valid, such an entry guard exists and our inductive inference routine can find it. The idea is to view safe switching states as positive examples and unsafe switching states as negative examples. The diagonally opposite corners of this hyperbox can then be found using binary search from the corners of the starting overapproximate hyperbox, assuming points in the hyperbox can be labeled as safe/unsafe (positive/negative). The search terminates when we have found the lower and upper diagonal corners as positive examples with their "immediate outer neighbours" as negative examples; for further details, see the hyperbox learning problem discussed by Goldman and Kearns [17].

  The positive/negative labels on states, required by the inductive routine, are generated by a deductive engine, as described below.
- *Deductive Reasoning:* In order to label a switching state s for a mode m as safe or unsafe, we need a procedure to answer the following question: if we enter m in state s and follow its dynamics, will the trajectory visit only safe states until some exit guard becomes true?

  This is a reachability analysis problem for purely continuous systems modeled as a system of ordinary differential equations (ODEs) with a single initial condition. This problem is known to be undecidable in general [44].

  However, in practice, this reachability problem can be solved for many kinds of continuous dynamical systems (including the intra-mode dynamics for the example shown in Fig. 8) using state-of-the-art techniques for *numerical simulation* (see, e.g., [43]). Thus, the deductive engine in our sciductive approach is a numerical simulator that can handle the dynamics in each mode of the multi-modal dynamical system. The numerical simulator must be *ideal*, in that it must always return the correct YES/NO answer to the above reachability question.

  The reader might wonder why a numerical simulator is termed as a deductive engine. Indeed, on the surface a numerical simulator seems quite different from a deductive theorem prover. However, on closer inspection one finds that both procedures employ similar deductive reasoning: they both solve systems of constraints using axioms about underlying theories and rules of inference, and they both involve the use of rewrite and simplification rules.

[4] Recall that a hyperbox corresponds to a conjunction of interval constraints over the continuous variables. The requirement for the vertices of the hyperbox to lie on a discrete grid is equivalent to requiring the constant terms in the hyperbox to be rational numbers with known finite precision.

Our overall approach to switching logic synthesis for safety properties [20] operates within a fixpoint computation loop that initializes each guard with an overapproximate hyperbox, and then iteratively shrinks entry guards using the hyperbox learning algorithm that selects states, queries the simulator for labels, and then infers a smaller hyperbox from the resulting labeled states.

5.3 Theoretical Guarantees

If the structure hypothesis is valid and we have an ideal numerical simulator, our approach to switching logic synthesis for safety properties [20] is *sound and complete*. This follows from three aspects: (i) the initialization of each guard with an *overapproximate hyperbox*; (ii) the structure hypothesis that ensures that the safe switching states in each iteration will form a hyperbox, and (iii) the learning algorithm that yields a hyperbox for each guard at each iteration that excludes all negative examples (unsafe switching states) and includes all positive examples (safe switching states).

However, if the structure hypothesis does not hold, or if the numerical simulator is non-ideal, then our approach cannot be guaranteed to be sound or complete. For this reason, if one cannot prove or otherwise reasonably assume the structure hypothesis to hold for the class of systems of interest, and the simulator to be ideal, then one must separately formally verify that the synthesized system satisfies the safety property. The numerical simulator could also be replaced by an alternative reachability oracle.

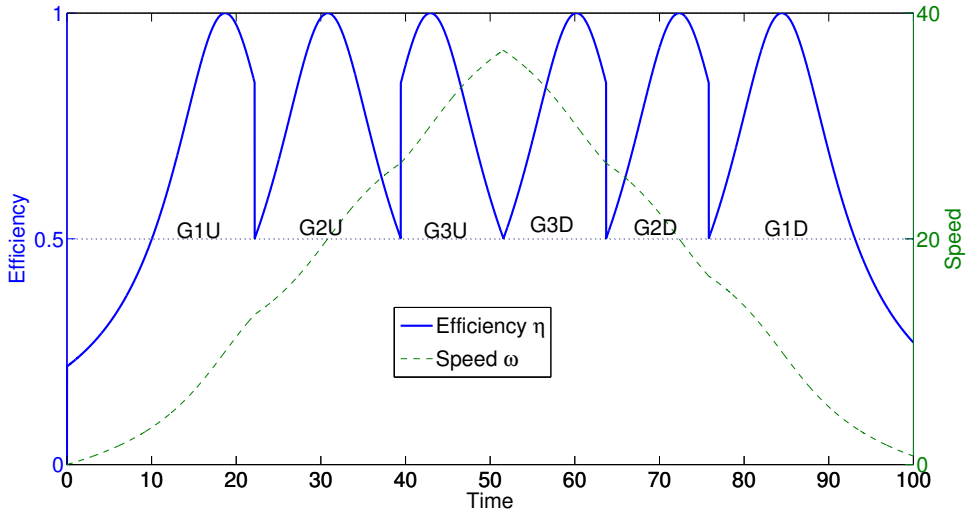

Figure 9: Transmission efficiency and speed with changing gears

5.4 Sample Result

We present here a sample result obtained for the automatic transmission example of Fig. 8. Our procedure combines learning hyperboxes (intervals, in this example) with a Matlab-based numerical simulator within an overall fixpoint computation loop. The outer loop starts with all guards set to the safety property ϕ_S (defined above) and iteratively shrinks each guard using the binary search-based inductive procedure described above.

The final set of guards obtained after fixpoint computation are as follows.

$$\begin{aligned} g_{N1U}, g_{11U} &: 0 \leq \omega \leq 16.70 \\ g_{12U}, g_{22U} &: 13.29 \leq \omega \leq 26.70 \\ g_{23U}, g_{33U} &: 23.29 \leq \omega \leq 36.70 \ , \ g_{33D} : 23.29 \leq \omega \leq 36.70 \\ g_{32D}, g_{22D} &: 13.29 \leq \omega \leq 26.70 \\ g_{21D}, g_{11D} &: 0 \leq \omega \leq 16.70 \ , \ ; \ g_{1ND} : \theta = \theta_{max} \wedge \omega = 0 \end{aligned} \tag{1}$$

Slightly modifying the safety property to require that the system spends at least 5 seconds in each of the six gear modes, we obtain the following modified set of guards:

$$\begin{aligned} g_{N1U} &: \omega = 0 \ , \ g_{11U} : \omega = 0 \\ g_{1ND} &: \theta = \theta_{max} \wedge \omega = 0 \ , \ g_{12U} : 13.29 \leq \omega \leq 23.42 \\ g_{11D} &: 1.31 \leq \omega \leq 16.70 \ , \ g_{23U} : 26.70 \leq \omega \leq 33.42 \\ g_{22D} &: \omega = 26.70 \ , \ g_{33D} : \omega = 36.70 \\ g_{32D} &: 16.58 \leq \omega \leq 26.70 \ , \ g_{33U} : 23.29 \leq \omega \leq 33.42 \\ g_{21D} &: 1.31 \leq \omega \leq 16.70 \ , \ g_{22U} : 13.29 \leq \omega = 23.42 \end{aligned} \tag{2}$$

The plot of the behavior of the transmission system when it is made to switch from Neutral mode through the six gear modes and back to the Neutral mode is shown in Figure 9. The efficiency η is always greater than 0.5 when the speed is higher than 5 and we spend atleast 5 seconds in the six gear modes. Starting from $\theta = 0, \omega = 0$, the synthesized system reaches $\theta = \theta_{max}$ with $\omega = 0$.

6 Conclusions and Future Directions

This paper posits that sciduction, a tight integration of induction and deduction with structure hypotheses, is a promising approach to addressing challenging problems in formal verification and synthesis. We have demonstrated some initial results in this regard, summarized in Table 1.

Application	$\mathcal{H}$	I	$\mathcal{D}$
Timing analysis (Sec. 3)	$w+\pi$ model & constraints	Game-theoretic online learning	SMT solving for basis path generation
Program synthesis (Sec. 4)	Loop-free programs from component library	Learning from distinguishing inputs	SMT solving for input/program generation
Switching logic synthesis (Sec. 5)	Guards as hyperboxes	Hyperbox learning from labeled points	Numerical simulation as reachability oracle

Table 1: **Three Demonstrated Applications of Sciduction.** For each application, we briefly describe the structure hypothesis $\mathcal{H}$, the inductive inference engine I, and the deductive procedure $\mathcal{D}$.

We conclude with some thoughts on further work on the sciductive approach and its applications.

Structure Hypothesis Testing/Verification. Recall that the soundness guarantees of sciduction only hold when the structure hypothesis is valid. A limitation of the current demonstrations of sciduction is that we currently do not have a systematic and general approach for checking the validity of the structure hypothesis. For example, in the program synthesis application of Sec. 4, how can we be sure that the library of components is sufficient to synthesize the program? As noted in Fig. 6, if the structure hypothesis does not hold, it

is possible to output an incorrect program. In this case, testing the structure hypothesis requires checking equivalence of the generated program against the specification, which may be expensive. More effective and generally-applicable methods for testing the structure hypothesis are required.

Integrating Induction and Deduction. Sciduction offers ways to integrate inductive reasoning into deductive engines, and vice-versa. It is intruiging to consider if SAT and SMT solvers can benefit from a sciductive approach — for example, using inductive reasoning to guide the solver for specific families of SAT/SMT formulas. Similarly, how can one effectively use deductive engines as oracles in learning algorithms? Are there new concept learning problems that can be effectively solved using this approach?

New Applications. An interesting direction is to take problems that have classically been addressed by purely deductive methods and apply the sciductive approach to them. For example, consider the problem of synthesis from LTL specifications. One practical challenge for this problem is in writing complete and consistent specifications, of which the environment assumptions are a large part. In recent work, we have demonstrated that environment assumptions can be mined from traces and counter-strategies [25]. It would be interesting to see if the synthesis algorithms themselves can be made more scalable using sciduction.

Sciduction can be used in generating abstractions or inductive invariants for verification. For example, we have recently used a combination of induction on decision trees (see [34]) and SMT-based ("term-level") model checking using UCLID [10] to perform conditional term-level abstraction of hardware designs [8]. Much remains to be explored in this area.

Controller synthesis for hybrid systems also remains an important domain with several applications. We have obtained some initial results on synthesizing switching logic for optimality, rather than just safety [21].

Another direction is to generalize the ideas used for timing analysis to other quantitative properties of cyber-physical systems, and also for verification problems at the hardware-software interface ("hardware-software verification"). In both settings, generating environment models can be quite challenging, and, from our experience with timing analysis, it appears that sciduction can be effectively brought to bear on these problems.

Acknowledgments

This article is a result of ideas synthesized and verified (!) over the last few years in collaboration with several great students and colleagues. In particular, Susmit Jha is a major contributor to this work, especially to Sections 4 and 5, which are part of his Ph.D. thesis research. Other collaborators include Jonathan Kotker and Alexander Rakhlin (Section 3), and Sumit Gulwani and Ashish Tiwari (Sections 4 and 5). This paper has benefited from feedback on talks on this work given by the author at several venues in 2009-11, including Princeton University, UC Berkeley, University of Texas at Austin, University of Pennsylvania, Carnegie Mellon University, Stanford University, IIT Bombay, FMCAD 2010, and the 2010 Gigascale Systems Research Center (GSRC) annual meeting.

The research reported here has been supported in part by several sponsors including the National Science Foundation (CNS-0644436, CNS-0627734, and CNS-1035672), Semiconductor Research Corporation (SRC) contracts 1355.001 and 2045.001, an Alfred P. Sloan Research Fellowship, the Hellman Family Faculty Fund, and the Gigascale Systems Research Center (GSRC) and MultiScale Systems Center (MuSyC), two of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity.

References

[1] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, February 1995.

[2] Dana Angluin. Queries and concept learning. *Machine Learning*, 2:319–342, 1988.

[3] Dana Angluin and Carl H. Smith. Inductive inference: Theory and methods. *ACM Computing Surveys*, 15:237–269, September 1983.

[4] Eugene Asarin, Olivier Bournez, Thao Dang, Oded Maler, and Amir Pnueli. Effective synthesis of switching controllers for linear systems. In *Proceedings of the IEEE*, volume 88, pages 1011–1025, 2000.

[5] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)*, pages 203–213, June 2001.

[6] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 4, chapter 8. IOS Press, 2009.

[7] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. *Formal Methods in System Design*, 18(2):141–162, 2001.

[8] Bryan Brady and Sanjit A. Seshia. Learning conditional abstractions. Technical Report UCB/EECS-2011-24, EECS Department, University of California, Berkeley, April 2011.

[9] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[10] Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In E. Brinksma and K. G. Larsen, editors, *Proc. Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 78–92, July 2002.

[11] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pages 52–71, 1981.

[12] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *12th International Conference on Computer Aided Verification (CAV*.

[13] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.

[14] J.E.R. Cury, B.A. Krogh, and T. Niinomi. Synthesis of supervisory controllers for hybrid systems based on approximating automata. *IEEE Transactions on Automatic Control*, pages 564–568, 1998.

[15] Dimitra Giannakopoulou and Corina S. Pasareanu, eds. Special issue on learning techniques for compositional reasoning. *Formal Methods in System Design*, 32(3):173–174, 2008.

[16] Harold Fox. *Agent problem solving by inductive and deductive program synthesis*. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 2008.

[17] Sally A. Goldman and Michael J. Kearns. On the complexity of teaching. *Journal of Computer and System Sciences*, 50:20–31, 1995.

[18] Anubhav Gupta. *Learning Abstractions for Model Checking*. PhD thesis, Computer Science Department, Carnegie Mellon University, 2006.

[19] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 215–224, 2010.

[20] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Synthesizing switching logic for safety and dwell-time requirements. In *Proceedings of the International Conference on Cyber-Physical Systems (ICCPS)*, pages 22–31, April 2010.

[21] Susmit Jha, Sanjit A. Seshia, and Ashish Tiwari. Synthesizing switching logic to minimize long-run cost. *CoRR*, abs/1103.0800, 2011.

[22] Matt Kaufmann, Panagiotis Manolios, and J. Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[23] T. John Koo, George J. Pappas, and Shankar Sastry. Mode switching synthesis for reachability specifications. In *HSCC*, pages 333–346, 2001.

[24] Robert Kurshan. Automata-theoretic verification of coordinating processes. In Guy Cohen and Jean-Pierre Quadrat, editors, *11th International Conference on Analysis and Optimization of Systems – Discrete Event Systems*, volume 199 of *Lecture Notes in Control and Information Sciences*, pages 16–28. Springer Berlin / Heidelberg, 1994.

[25] Wenchao Li, Lili Dworkin, and Sanjit A. Seshia. Mining assumptions for synthesis. In *Proceedings of the Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, July 2011. To appear.

[26] J. Lygeros. Lecture notes on hybrid systems. 2004.

[27] Sharad Malik, Joao Marques-Silva, and Ines Lynce. Conflict-driven clause learning sat solvers. In Armin Biere, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 1, chapter 4. IOS Press, 2009.

[28] Sharad Malik and Lintao Zhang. Boolean satisfiability: From theoretical hardness to practical success. *Communications of the ACM (CACM)*, 52(8):76–82, 2009.

[29] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM TOPLAS*, 2(1):90–121, 1980.

[30] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems:Specification*. Springer-Verlag, 1992.

[31] P. Manon and C. Valentin-Roubinet. Controller synthesis for hybrid systems with linear vector fields. In *IEEE International Symposium on Intelligent Control*, pages 17–22, 1999.

[32] Kenneth L. McMillan. Interpolation and SAT-based model checking. In *Proc. 15th International Conference on Computer-Aided Verification (CAV)*, pages 1–13, July 2003.

[33] Kenneth L. McMillan. Relevance heuristics for program analysis. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 145–146. ACM Press, 2008.

[34] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[35] T. Moor and J. Raisch. Discrete control of switched linear systems. In *European Control Conference*, 1999.

[36] Stephen Muggleton and Luc de Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19-20(1):629–679, 1994.

[37] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. In *7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3855 of *Lecture Notes in Computer Science*, pages 364–380. Springer, 2006.

[38] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 179–190, 1989.

[39] Phillip Porras, Hassen Saidi, and Vinod Yegneswaran. An analysis of conficker's logic and rendezvous points. Technical report, SRI International, March 2009.

[40] Wei Qin and Sharad Malik. Simit-ARM: A series of free instruction-set simulators and micro-architecture simulators. http://embedded.eecs.berkeley.edu/mescal/forum/2.html.

[41] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, number 137 in LNCS, pages 337–351, 1982.

[42] Reinhard Wilhelm et al. The Determination of Worst-Case Execution Times—Overview of the Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 2007.

[43] Jaijeet Roychowdhury. Numerical simulation and modelling of electronic and biochemical systems. In *Foundations and Trends in Electronic Design Automation*, volume 3, pages 97–303. 2009.

[44] Keijo Ruohonen. Undecidable event detection problems for ODEs of dimension one and two. *Informatique Théorique et Applications (ITA)*, 31(1):67–79, 1997.

[45] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010.

[46] Sanjit A. Seshia. Quantitative analysis of software: Challenges and recent advances. In *Proc. Formal Aspects of Component Software (FACS)*, 2010.

[47] Sanjit A. Seshia and Jonathan Kotker. GameTime: A toolkit for timing analysis of software. In *Proc. Tools and Algorithms for the Analysis and Construction of Systems (TACAS)*, 2011.

[48] Sanjit A. Seshia and Alexander Rakhlin. Game-theoretic timing analysis. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 575–582. IEEE Press, 2008.

[49] Sanjit A. Seshia and Alexander Rakhlin. Quantitative analysis of systems using game-theoretic learning. *ACM Transactions on Embedded Computing Systems (TECS)*, 2011. To appear.

[50] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodík, Vijay A. Saraswat, and Sanjit A. Seshia. Sketching stencils. In *PLDI*, pages 167–178, 2007.

[51] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.

[52] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.

[53] Saurabh Srivastava, Sumit Gulwani, Swarat Chaudhari, and Jeffrey S. Foster. Path-based inductive synthesis for program inversion. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011. To appear.

[54] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 313–326, 2010.

[55] Symantec Corporation. Internet security threat report volume XIV. http://www.symantec.com/business/theme.jsp?themeid=threatreport, April 2009.

[56] P. Tabuada. Controller synthesis for bisimulation equivalence. *Systems and Control Letters*, 57(6):443–452, 2008.

[57] Claire J. Tomlin, John Lygeros, and S. Shankar Sastry. A game theoretic approach to controller design for hybrid systems. In *Proceedings of the IEEE*, volume 88, pages 949–970, 2000.